IMSE Milestone 3

**Reasons for designing the NoSQL database**

What we know that a relational database is not always the rightest choice to use in a system. Our system is based on renting cars, which means every day we gain more costumers, more cars and with all this, more rents which includes bills and insurances for each of them. Therefore, we need more space to store all this data. What we want is that a larger system which can analyse the customer activity and tracks our cars rental status. We should be able to see these statically and adapt the new cars in the system. Generally, as a rental company, we want to be prepared for a big amount of data and dare to switch the system as early as possible.

**Why to use NoSQL documents and Collections**

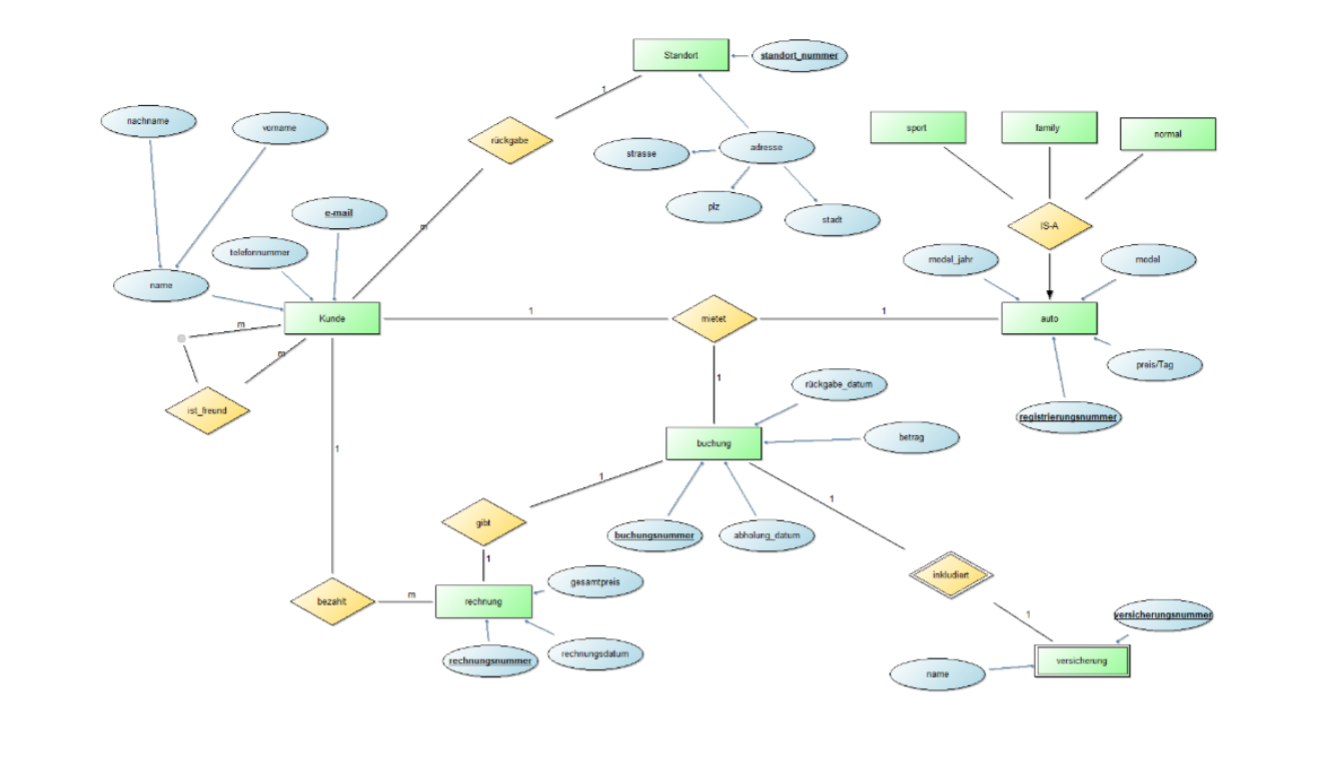
***The document*** is the unit of storing data in a MongoDB database and it uses JSON format for storing data, which is a lightweight, thoroughly explorable format used to interchange data between various applications. Documents are analogous to the records of an RDBMS. We can show the difference as following:

|  |  |
| --- | --- |
| **RDBMS** | **MongoDB** |
| Table | Collection |
| Column | Key |
| Value | Value |
| Records/Rows | Document/Object |

Insert, update, and delete operations can be performed on a collection***. A collection*** may store several documents. A collection is analogous to a table of an RDBMS and it may store documents those who are not same in structure. This is possible because MongoDB is a Schema-free database (We describe schema as the structure of data in RDMS). In the other hand MongoDB doesn’t require such a set of formula defining structure of data.

**Design**

Before we created our RDMS we designed our renting database as you can see in the ER-Diagram below:



From this diagram, we decided the relations for our NoSQL database as following:

* Booking <-> insurance One-To-One
* Booking <-> Car One-To-Many (few)
* Customer <–> Costumer: Many-To-Many
* Customer <-> Location: One-To-Many
* Booking <–> Bill: One-To-One
* Costumer <–> Bill: One-To-Many (actually many)

**Booking <–> Insurance: One-To-One**

An insurance is depending on a booking, which means if a booking exists that a unique insurance is creating for the specific booking. If there are 0 bookings in the DB than we cannot show an insurance. The collection “Booking” should include insurance. Therefore, we decided the correct relation would be One-To-One.

**Costumer <–> Bill: One-To-Actually Many**

We know that one costumer can rent many cars separately. For this reason, it would be a good way if we store the bills in the Costumers. With this relation we will be able to see all bills generated for that costumer. But we should also consider that this kind of a big data need more space in the disk and cost much more time if we update/search it.

**Booking <–> Car: One-To-A-Few**

With a booking, it is possible to rent a few cars. We decided to use one-two-a-few relation in order to determine which cars are rented with a specific bill. For that, we should store carIDs as a list (Array) in Booking. A little issue here would be a little more storage place in the disk.

**Costumer <–> Costumer: Many-To-Many**

Costumers can be friends with each other. In order to extend our business, we want to use these friendships to gain more costumers and we want to offer our already existed costumer 10% discount for every costumer they bring. Therefore, we should remember this relation in the system and we decided to use here a reference of the costumers to find out which costumer is friends with each other.

**Costumer <–> Location: One-To-Many**

Costumers locate the car which they rented in different locations. In our systems there are already various locations and we want to store this information in order to calculate the ratio of a costumer, that shows us their favourite locations or their preferred locations to hand over the car. With this, we can analyse the most preferred locations and we will be able to show these statically in the system.

**Booking <–> Bill: One-To-One**

After a costumer booked a car/cars, it generates a bill automaticlly. For each booking, it created a bill, which includes the total price. Thus, our choice is to use one-to-one relation in this case.

**Inherited Relations (Sport-, Normal-, Family car)**

We made our decisions that we store the cars attributes in each inherited entity.

**Our Sharding Strategy: Hashed Sharding**

As we mentioned before, we want to expand our company and with using NoSQL Database, Sharding became even more important. The advantages of this strategies are followings:

* Monotonically Changing Values
* High Shard Key Frequency which supports compound keys
* Chunk size can be configured via Hash function

With this strategy, we have to use a hast function and every collection is stored with a key in the system, which makes it possible to search or update a document really fast.